January 1995

Architecture and Performance of the Mether Network Shared Memory

John H. Shaffer
University of Pennsylvania

Ronald G. Minnich
Supercomputing Research Center

Jonathan M. Smith
University of Pennsylvania, jms@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-13.

Architecture and Performance of the Mether Network Shared Memory

Abstract

Mether is a Network Shared Memory (NSM). It allows applications on autonomous computers connected by a network to share a segment of memory.

NSMs offer the attraction of a simple abstraction for shared state, i.e., shared memory. NSMs have a potential performance problem in the cost of remote references, which is typically solved by *grouping* memory into larger units such as pages, and *caching* pages. While Mether employs grouping and caching to reduce the average memory reference delay, it also removes the need for many remote references (page faults) by providing a facility with relaxed consistency requirements.

Applications ported from a multiprocessor supercomputer with shared memory to a 16-workstation Mether configuration showed a cost/performance advantage of over 300 in favor of the Mether system. While Mether is currently implemented for Sun-3 and Sun-4 systems connected via Ethernet, other characteristics (such as a choice of page sizes and a semaphore-like access mode useful for process synchronization) should suit it to a wide variety of networks. A reimplementation for an alternate configuration employing packet-switched networks is in progress.

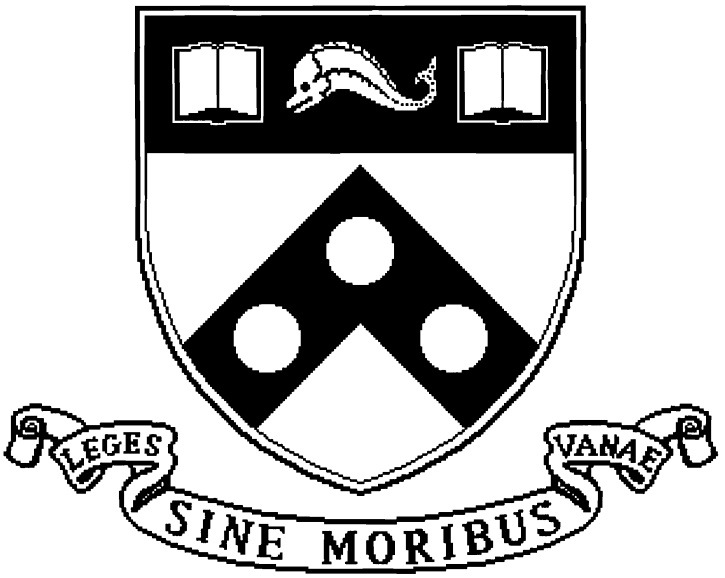
Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-95-13.

Architecture and Performance of the Mether Network Sharted Memory

MS-CIS-95-13

John H. Shaffer
Ronald G. Minnich
Jonathan M. Smith



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

April 1995

Architecture and Performance of the Mether Network Shared Memory

John H. Shaffer
Distributed Systems Lab
University of Pennsylvania
Philadelphia, PA

Ronald G. Minnich
Supercomputing Research Center
Bowie, MD

Jonathan M. Smith
Distributed Systems Lab
University of Pennsylvania
Philadelphia, PA

Abstract

Mether is a Network Shared Memory (NSM). It allows applications on autonomous computers connected by a network to share a segment of memory.

NSMs offer the attraction of a simple abstraction for shared state, i.e., shared memory. NSMs have a potential performance problem in the cost of remote references, which is typically solved by *grouping* memory into larger units such as pages, and *caching* pages. While Mether employs grouping and caching to reduce the average memory reference delay, it also removes the need for many remote references (page faults) by providing a facility with relaxed consistency requirements.

Applications ported from a multiprocessor supercomputer with shared memory to a 16-workstation Mether configuration showed a cost/performance advantage of over 300 in favor of the Mether system. While Mether is currently implemented for Sun-3 and Sun-4 systems connected via Ethernet, other characteristics (such as a choice of page sizes and a semaphore-like access mode useful for process synchronization) should suit it to a wide variety of networks. A reimplementation for an alternate configuration employing packet-switched networks is in progress.

1 Introduction

Virtual Memory support (VM) in modern computer architectures has allowed a number of useful innovations in software architectures to support applications. Among these are memory-mapped files, flexible use of multiple paging devices, copy-on-write semantics and application-defined pagers. Applications using virtual memory can share a segment of memory by mapping portions of their virtual address space to a common area of real memory. This mapping strategy has traditionally been limited to a single machine to keep the translation of the mapping simple, fast, and consistent. "Consistent" means that **read** operations return the value of the most recent **write** operation.

A Network Shared Memory (NSM) is a memory space that is logically shared by a distributed application on distinct computers connected by a communication network. It uses VM support to provide the illusion that the data are actually shared as in the local case. Figure 1 illustrates this concept of "shared state". Communication itself is an artifact of sharing some subset of one

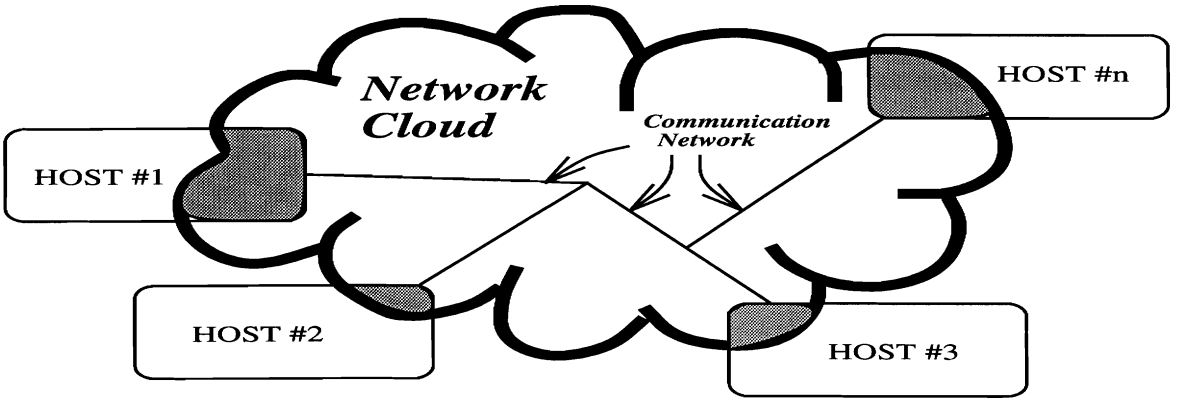


Figure 1: "Shared state" in an NSM.

processes' state space with another process. Network Shared memory is an elegant fit into this paradigm.

1.1 Distributed Programming Paradigms

A distributed system[38],[15] is a group of computers cooperating with each other to achieve some goal. These computers are autonomous, in that each computer has an independent flow of control. We assume there is no physical sharing of memory among computers[21], [3]. Processes running on different computers have distinct address spaces. They communicate by sending and receiving data encapsulated as *messages*. Message-passing primitives are then used by applications to communicate with cooperating computers. One necessary characteristic of cooperation is some form of state (data) sharing[9].

Unfortunately, the message passing abstraction does not support data sharing directly. Data sharing is still possible with message passing, by maintaining the shared data in a dedicated process and operating on the data by sending operators to this process[26]. Other methods may involve moving data around explicitly using message passing primitives. Spector[36], for example, observed that remote references might offer a better model for efficient communications.

Remote procedure call (RPC)[5], was introduced to provide a procedure-call like communications interface. Since the "procedure call" is executed in a separate address space, it is difficult for the caller to pass context related data or complicated data structures, i.e., parameters must be passed by value. RPC designers indicated the desire for distributed shared memory so that data could be passed by reference. RPC can be viewed as a half-step towards shared memory, since the semantics are basically those of shared memory. The major limitations are imposed by implementation constraints (e.g., limited copying of complex data structures).

1.2 Latency and Shared Address Spaces

A shared memory space supports data sharing with very little overhead (and hence communication). [17][11][36] However, there are technical difficulties in providing complete shared memory semantics in a decentralized setting.

A major impediment to high performance is the "latency" of remote references. "Latency" is the time required for a memory reference and can be represented as a ratio of remote access time to and local access time. If the value of this ratio is large, the mismatch must be remedied for "performance transparency"[18]. Ideally, processes on each node should be able to access the same address space with fetch and store operations[32]. However, since the latency of communication through the network may be high, simple implementation of the fetch and store as remote operations to a shared memory server may not be attractive[36].

Several models of the shared address space have been investigated by earlier research. A DSM can be an unstructured ("flat") and paged virtual address space [25], a segmented single level store [33][8], an object-oriented model visible to programming language syntax and semantics[2], or even a physical address space [12].

A difficulty any shared memory system must face is maintaining consistent shared state. Succinctly, "consistency" is a rule for the semantics with which a reference to a variable (i.e. its name) is resolved to a value. Typically[1], the semantics are these: when the object's name is used to obtain its value, the *most recently written* value is returned. This has implications for performance in a distributed system, since (1) the most recently written value must be located, and (2) in an implementation where caching of shared values is used, "old" copies must be replaced or invalidated. Li and Hudak's [25] work focused on efficient algorithms for maintaining coherent state in DSMs, and most other work has maintained this model.

The desire for performance transparency has led to a number of proposals for reducing the *average* latency per remote reference, such as:

- Pre-fetching of pages or objects, in anticipation of their use [35].
- Pre-sending of pages or objects, in anticipation of their use [39].
- Using the network fabric itself to locate and store data [37].

1.3 Relaxing Consistency

Another approach to addressing NSM performance is to reduce the number of communications (and hence the latency) by relaxing the consistency semantics. This approach has three major advantages: First, it can significantly enhance performance even when the limits of optimizing

remote page-fault latencies have been reached. Second, since consistency-maintenance becomes more difficult with increasing numbers of participants, it would tend to scale better. Third, more robustness in the face of dropped messages is achieved.

However, the approach suffers from an equally major disadvantage: not all applications can tolerate inconsistent state - in fact, it is unclear how to characterize applications which can. However, the existence of such applications has been observed before[22]. One early proposal for a memory-model network supporting different modes of consistency was *problem-oriented shared memory*[9].

In this system, an inconsistent memory was proposed, but the mechanism for update of inconsistent data was left undefined. Lipton, et. al.'s PRAM system[27] has non-traditional consistency semantics. However, as in problem-oriented shared memory, it is not possible to control the consistency semantics of the memory. Clouds[33] allows processes to use "inconsistent" and "consistent" memory; the mechanism for switching back and forth is via a system call. Work by Zwaenepoel, et al. [4] has focused on implementation of software support for inconsistency. Hutto[20] has proposed semantics and formalism with which inconsistency can be discussed. Minnich[30] examined architectural issues in software systems providing relaxed consistency constraints for selected applications.

1.4 Organization of this Paper

In this paper, a software architecture for supporting applications with relaxed consistency constraints is presented. To test the architecture and the assumptions embedded in it, applications were ported from a Cray-2 shared memory multiprocessor to Mether. In the body of the paper, Section 2 describes the Mether model followed by details of more Mether capabilities in Section 3. Data structures and functional divisions are described in Section 4. Finally, Section 5 discusses application performance and Section 6 concludes the paper.

2 The Mether Model

2.1 Mether Application Interface

There are a number of models and interface paradigms with which the behavior of a networked shared memory can be offered to applications programmers. The entirety of an application's address range could be shared, requiring explicit declaration of, or negotiation for, "private" address ranges. This suffers from both a lack of robustness for conventional applications, and a need for programmers to be aware of distributed resource allocation semantics. A distinguished range of addresses could be allocated automatically for each process in a distributed application. This eases

setup, but may penalize applications uninterested in the feature. Applications could use specialized access primitives (e.g., a shared memory LOAD and STORE) to access shared memory, but this adds a considerable, and unavoidable, performance penalty for each DSM access[26].

The strategy we have chosen is for each process in an application to explicitly *assign* a set of "well-known" NSM addresses to an area of its private address space. In this way, applications can choose to include the feature, and customize its access through a range of addresses as best suits requirements. Mether does this by providing an interface equivalent to a programming language's memory allocator - the pointer returned can be used in the same manner. A small code fragment, given in Figure 5 (analyzed in Section 5.1.1) illustrates the usage.

Mether is implemented within the UNIX operating system. UNIX models objects as named entries in the file system name space; system calls allow objects to be created, accessed, and controlled. Mether's namespace entry is */dev/mether*, which is a UNIX "special file", providing access to non-file semantics (such as remote page-fault resolution) through file system operations such as *open()*, *close*, *select()*, and *ioctl()*, and operations such as *mmap()* which meld memory management and the file system. *Open()* allows access to the Mether functionality through a file system name space entry, and *close()* disallows such access. *Mmap()* associates a process address range with a range of addresses in the Mether address space. *Ioctl()* controls driver parameters and the states of individual pages, and *select()* allows process synchronization based on events such as Mether page faults. Detailed parameters for Mether's *ioctl()* interface are given in [30]. The Mether library builds a higher-level interface for use by applications programmers using these system entry points as a basis.

Mether is currently implemented in two parts (See Figure 4), an operating system kernel page manager (the features accessed via system calls) and a user-level server. The page manager is built as a device driver in the operating system; the user-level server runs as a program and is responsible for communications. We will describe the system as seen by a programmer, and then describe the components.

2.2 Application-directed Semantics

One of the major challenges of designing a system is selecting features and parameterizing their behavior; the difficulty stems from trying to anticipate the applications' needs. On the one hand, a single value, such as a page size, can be chosen in an attempt to simplify the system. However, choosing such a value can be difficult, and susceptible to poor anticipation of application-specific tradeoffs. On the other hand, a flexible parameterization of tradeoffs and features can be offered. The strength of this approach is the ability of applications to precisely customize the system's

behavior to their needs. Weaknesses stem from three facts: most applications tend to customize system behavior poorly, the complexity required for such flexibility is high, and poor performance is often a consequence of generality.

Mether chooses a middle ground, offering exactly two choices for each of three selected parameters: page size, process synchronization, and consistency semantics. A choice of *page sizes* allows the observed dichotomy between small control packets and large transport packets to be exploited[7]. A choice of *process synchronization* (or more accurately, process blocking) allows applications to poll for, or wait for, updates to the shared memory. Finally, a choice of *consistency semantics* between strong consistency, which requires writes to be immediately visible to readers, and weak consistency, which requires writes to be eventually visible to readers, allows applications to increase performance by reducing their page faulting rate.

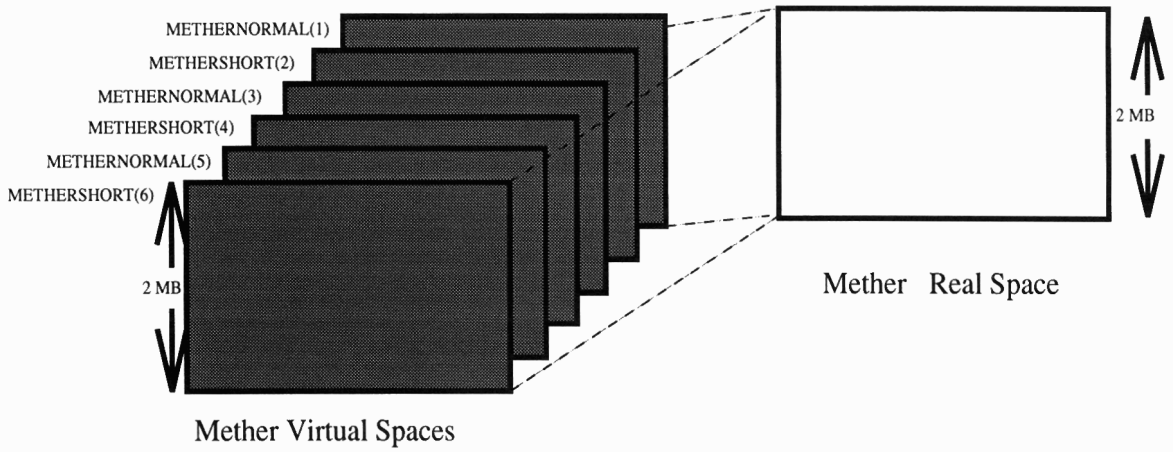


Figure 2: How `methersetup` maps Mether pages.

Mether provides six (sub)address spaces, corresponding to the parameter choices described above. They are:

1. *Strongly consistent, 8192-byte page, demand-driven page faults*
2. *Strongly consistent, 32-byte page, demand-driven page faults*
3. *Weakly consistent, 8192-byte page, data-driven page faults*
4. *Weakly consistent, 32-byte page, data-driven page faults*
5. *Weakly consistent, 8192-byte page, demand-driven page faults*
6. *Weakly consistent, 32-byte page, demand-driven page faults*

The 8192-byte pages are called full sized pages; the 32-byte pages are called short pages. The user specifies in the mmap call whether the data is to be accessed as read-only or writeable, which determines whether the inconsistent (read-only) or consistent (writeable) copy of the data is

requested on a page fault. The address spaces in Figure 2 determine the access mode for a reference. In fact, the underlying operating systems support code only provides four spaces; the `mether setup` function adds two more. The last two 2-megabyte address spaces are different from the first two only in that `mether setup` maps them in as read-only, whereas the first two are writeable.

A global view of the various Mether address spaces is shown in Figure 3.

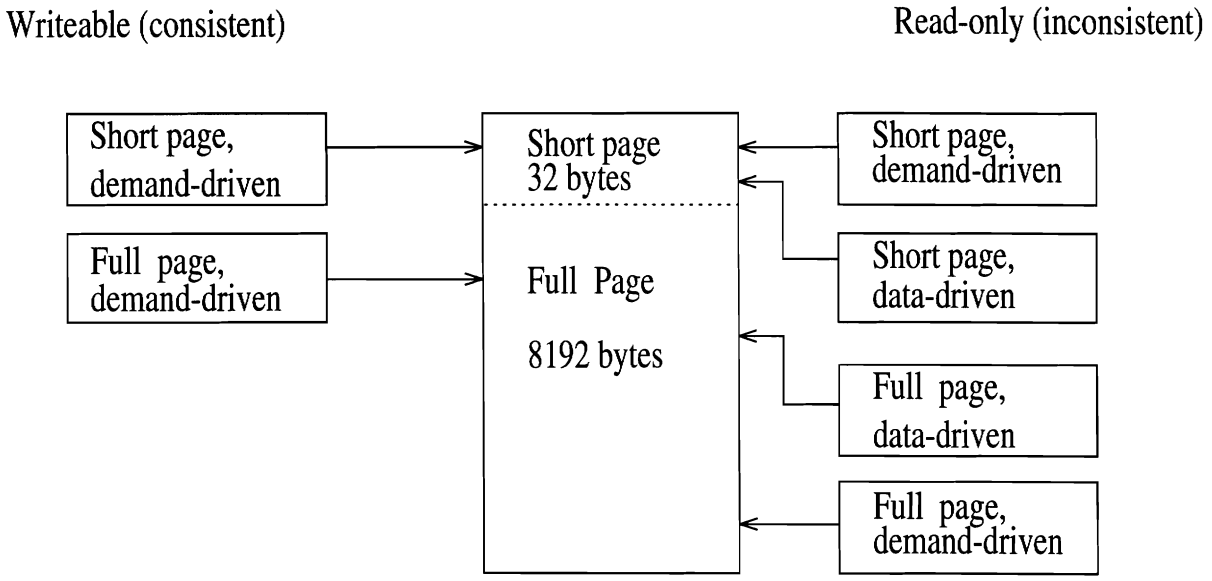


Figure 3: The Mether address space

Figure 3 notes:

1. *The choice of the read-only space or the writeable space is chosen when the application maps the Mether address space in.*
2. *Note that the consistent space can only be demand-driven.*
3. *The choice of full or short page, demand or data driven is determined by two address bits in the Mether address space.*
4. *If further applications demand it, we may opt for four different page sizes- one more bit of address.*

3 Additional Semantic Choices in the Mether Memory Model

In its most basic form Mether is identical to systems described in [23], [16], [24], and [12]. These systems provide a shared memory model to programs that is identical to a shared memory multiprocessor. This model is often called a strongly coherent memory model.

Although parallel program semantics for readers and writers are preserved when the strongly coherent memory model is enforced, in these implementations, comparable performance can not be maintained. Systems which use a network to effect movement of data inevitably have a higher average and worst-case latency than a multiprocessor, in some cases up to five orders of magnitude higher.

In [31] and in [30], the implications on applications of this higher latency were studied. We measured the effect of network latency on the operation of Mether Version 1, which provided a strongly coherent memory model, and we determined that latency was a factor which could not be ignored in a networking environment. The measurements and their implications are applicable to other systems such as MemNet[12], Dash[14] and Ivy[24].

Mether Version 2 provides an extended memory model to applications. The extensions allow programs to make effective use of the processors and network while minimizing difficulties due to latency. Mether presents users with a virtual address space partitioned into pages. An Mether operation on any part of a page applies to the whole page as well. For example, a reference to part of a page will cause a whole page to be fetched.

The operations that Mether supports on pages differ in several crucial ways from other NSMs. On a memory-fetch by memory-fetch basis, programs can determine the type of service needed for that fetch. The types of service supported are:

User-Managed Consistency A program may choose to access a *strongly consistent* copy of a page, in which case it will have the only *writable* copy; or it may choose to access an *inconsistent* memory, in which case it will have access to a replicated copy of the strongly consistent version of the page. Over time, inconsistent copies of pages are updated to a more recent version of the consistent copy. Operations are provided in Mether that allow user-level control of the update process. The update may happen because the holder of the one consistent copy wishes to update all replicates; or because the holder of a replicate determines that a new copy should be accessed. There is a default mechanism, related to page reclamation, which ensures that pages will never be more than 30 seconds out of date.

Different-size objects Programs can, on a fetch-by-fetch basis, choose to access pages which have a very low transmission cost but do not move much data (currently 32 bytes); or they may choose to access large pages which have a higher transmission cost but are useful for moving large amounts of data (currently 8192 bytes). The small pages ("short pages") are useful for synchronization and storing small objects; the large pages are useful for bulk data transfer. These sizes are well matched to the protocol data unit sizes most commonly seen in networking environments[10][7].

Event-Driven Memory Synchronization In Event-Driven Memory Synchronization (EDMS) one process can pause in the middle of a memory read (i.e. after the address has been issued but before the data cycle is complete) and will continue only after another process has taken some action. This synchronization mechanism[31] minimizes the network and host load and hence to minimize the contribution made to the overall latency by host and network load.

4 Mether Data Structures and Functional Divisions

Mether is divided into a kernel driver and a user level communications server. The former manages the in-memory pages and the latter allocates pages as they are needed. In addition, it maintains the state descriptors for these pages, called Mether Page Table Entries (MPTEs). As page state changes, via *mmap()* requests or via Mether *ioctl()* system calls, the MPTEs are modified to reflect the state changes. Any user level program can *mmap()* in the MPTE structures, examine them, and determine which pages are currently wanted by other user level processes.

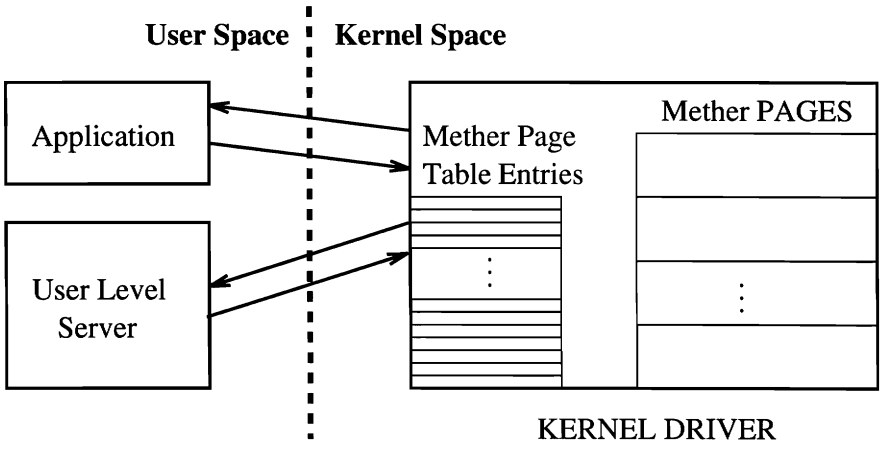


Figure 4: Functional setup of Mether.

When a page is not present on the local machine the user-level server will request it from the machine currently owning it. The user level server uses *mmap()* to map the MPTEs into its memory. Thus, the server sees a set of data structures with attributes that change over time; the change in these attributes drives the generation of page requests over the network. The server runs in an event-driven loop monitoring both network messages sent to it and changes in the Mether data structures.

The global address space of Mether may be much larger than any single system's main memory and therefore, a potential problem is the case when a page is unable to find a host with room for

it. In order to guard against this, every page in the Mether address space has a *home* host which *always* retains reserved storage for its assigned pages. This storage area is called *reserved memory* and is distributed among all the hosts on the system; typically each Mether host will have a "fair share" (i.e. on a system with 10 interfaces, 10%) of its memory as reserved storage, with the rest of storage available for other pages.

A page is created only from reserved space, and then only when referenced. When a non-reserved page is first referenced, a request for that page is generated by the user-level server. Only if that page is in some processor's reserved address space will space for it be allocated.

4.1 User Level Server

The user-level server (ULS) runs as an event-driven loop. It detects changes to page states and generates network messages as needed; it responds to network messages; and it effects changes in page states as needed.

4.2 User Level Server to User Level Server Communications

User level servers communicate with other ULS's in order to locate and transfer pages. There is also a limited amount of control information transferred between the ULSs.

The ULS's communicate in an environment displaying the following conditions:

- Unreliable transport. Rather than characterize it as unreliable, we will say that we expect to lose or duplicate a few packets in one hundred. In addition, when the network is heavily loaded, we may lose large groups of packets at once.

- No automatic retransmission for error control. Rather, if a client determines that a request has not been acknowledged in a reasonable time, another request will be sent. In fact no packet transmission has an acknowledge packet in the Mether protocols for any version of Mether.

- Transaction-oriented protocol. We treat page request/return activities as *transactions.*

To support the acknowledgment-free, transaction oriented model, we need a logical manner with which to distinguish transactions. A transaction must have a unique id that distinguishes it from all other transactions. This also implies that in the case of a request timeout, the new packets will carry the same transaction UID as the previous packets.

We generate unique ids via timestamps and source host IP addresses. The IP address is 32 bits. The timestamp is derived by taking the 64-bit system time and using the high-order eight bits of

the microseconds field and the low order 24 bits of the seconds field. This uniquely identifies a transaction to a four-millisecond interval in a four-year span, adequate for our use.

We also require that all the hosts using Mether have their time synchronized to at least the four millisecond resolution of the Mether timestamps. Note that this requirement is not difficult to meet if the systems concerned are running NTP[29]. This allows UIDs to be ordered.

Rather than being sent to a specific host, all packets are broadcast in the current implementation. As mentioned, this is not the case in the MNFS version which multicasts to the hosts with copies of the page in question.

4.3 ULS Communications with the User

Typically, the ULS runs as a daemon process. When the user wants to monitor or control the ULS it may be invoked interactively.

This interactive mode is useful both for debugging ULS to ULS communications and for testing changes made to the kernel driver. Thus, the ULS can be run in an interactive mode on a computer that does not support Mether and monitor Mether traffic on the network. This mode has been useful for generating trace references as input data for simulations.

5 Applications and Performance

Successful systems research is characterized by both *novelty* and *impact*, and the impact is gauged by the effect the system has on existing applications as well as its ability to stimulate new ones. The effect on applications can be argued in a number of ways. One is to analyze applications, identify the performance bottlenecks, possibly putting them into an applications "kernel", and show how the new system addresses these bottlenecks. The strength of the argument rests on the correct identification of the "kernel" of applications performance factors. Another method of argument is the use of existing applications which are migrated to the new system. The proof here is the demonstrated effect on real applications, although variations in application characteristics may make the results less general. We have used both methods to test Mether.

In this section we describe the behavior of Mether on a kernel utility, and three existing applications which have been modified to use Mether. By altering entire existing applications to use Mether rather than coding benchmarks, the utility and completeness of both the design and the implementation are validated.

The example "kernel" application is derived from a number of parallel programs requiring synchronization, and is structured as a simple producer-consumer problem. We analyze a code fragment including the Mether calls, and illustrate how the application's consistency and page size

selection affect the performance. We also show how these performance figures vary with the scale of the Mether configuration used.

The other three applications ported to Mether are:

- A *Monte Carlo program.* This application models a radiative heat transfer and uses Mether to store both problem parameters and results structures.
- A *sparse matrix solver.* This application uses Mether-based communications structures for communications and synchronization and a large shared array for storing results.
- A *DNA pattern matching algorithm.* This application uses Mether to store problem state, input data and results.

There were a number of other applications implemented using Mether, among which were a shared-memory emulation of Unix pipes which demonstrated that an Mether-based program could equal or exceed the performance of Unix pipes between two machines; an N-queens problem[1] which used shared memory structures for dispatching work and accumulating results; a 2-D graphics display which used a 4 by 4 grid of Sun ELCs to implement a 12-megapixel display; and a tree search problem. In the interest of brevity (and the fact that these applications do not show any further features of Mether) they will not be given further attention.

5.1 A Producer/Consumer Problem

Figure 5 shows a simple example of parallel programming using Mether.[2] The code gives an elementary solution to the well known producer/consumer problem. Recall that this problem requires some variation of a semaphore in order to coordinate two parallel processes. The Mether shared space is used for two purposes. One is to implement the semaphore and the other is to transfer data from the producer to the consumer. It is important to note that both control and data movement are done *via* Mether.

Each process, typically on different hosts, must first gain access to the Mether address space by opening the Mether device, which in turn initializes the address space.

5.1.1 Overview of Program

[Lines 1-5] Declaration of the `sentinel` structure which will be the manner of access to the Mether

[1]The N-queens problem gives a solution of how to orthogonally position vectors within N-space. An example is determining how one may position n queens on a chess board such that none is threatened by any other.

[2]Note: This example is intended to *introduce* the functionality of Mether.

Producer	Consumer
1 struct sentinel	1 struct sentinel
2 {	2 {
3 int syn, data;	3 int syn, data;
4 };	4 };
5 struct sentinel *sp;	5 struct sentinel *sp;
6 sp = methersetup();	6 sp = methersetup();
7 sp = METHERMAPCLASS(METHERBASE,	7 sp = METHERMAPCLASS(METHERBASE,
8 METHERNORMAL);	8 METHERNORMAL);
9 sp->syn = 0;	9
10 RunConsumer();	10
11 sp->data = 0;	11
12	12
13	13
14 while(sp->data < 1000)	14 while(sp->data < 1000)
15 {	15 {
16 sp->data = sp->data + 1;	16 while (sp->syn == 0)
17	17 ;
18 sp->syn = 1;	18
19 while (sp->syn == 1)	19 func(sp->data);
20 ;	20 sp->syn = 0;
21 }	21 }
22 exit();	22 exit();

Figure 5: The Producer and Consumer Problem

shared address space. The `syn` field is the synchronization variable and `data` the the conduit for process to process data transfer.

[Line 6] Open the Mether device and initialize the Mether system thus gaining access to Mether address space. (This function is provided in the Mether library.)

[Lines 7-8] Here the two processes are is gaining access to identical portions of Mether shared space since they both are mapping to `METHERBASE` which is the LOW section of memory. The `METHERNORMAL` instructs the system to get access through the normal sized (8K) pages and to use R/W access.

[Lines 9-11] `Producer` By setting `sp->syn = 0` a *write* is effected, initializing this variable, thus this page of shared space is swapped in. Same with `sp->data = 0`. Here we also start the `Consumer()` process. The order here is important because, as will be seen later, the `Consumer` waits on the `sp->syn` variable before it is able to continue.

[Lines 14-21, Producer] While `sp-> data < 1000` we increment and set `sp->syn =1`. Next is a simple spin-lock on `sp->syn`. It is now up to the `Consumer` to set `sp->syn` to zero before this spin-lock can be exited.

[Lines 14-21, Consumer] Here again, the main loop is performed 1000 times and then waits on `sp->syn` until it becomes a zero, performed by the `Producer`). The current value of `sp->data` is then used by function `func()` and upon return, resets `sp->syn=0` thereby allowing the `Producer` to continue.

5.1.2 Details of Operation

The processes are using the strongly consistent full size page (8192 bytes) Mether address space. The programmer creates a pointer to the first page of this address space using the METHERMAPCLASS macro. METHERMAPCLASS takes two parameters: a virtual address and an address space qualifier, usually one of the pre-defined constants whose names are shown in Figure 2. In this case, the programmer has chosen METHERBASE as the virtual address, which by convention is the address of the first page in the Mether address space; and the METHERNORMAL qualifier, which selects the strongly consistent full page size address space.

If the consumer or producer needs to access the `sp->syn` variable and it is not present, then an 8192 byte page must be moved across the network. While the page is in transit both processes are blocked and while the producer is testing the variable the consumer can not write to it. The interference between these two processes can become significant and is quantitatively discussed in [31] along with the way in which Mether can be used to eliminate it.

5.1.3 Optimizations Using Mether

One method with which this interference can be eliminated is for the processes to use the Mether *inconsistent* memory modes. Each process can continuously test (the spin-lock) the inconsistent copy of the variable, hence it is merely accessing a *copy* of the page in which the variable resides, *not* the *writable or consistent* page.[3] Now the consistent copy of a page is available only to the process that needs write access. The interference caused by migration of the consistent page will now be eliminated.

The problem with using this approach alone is that, although it reduces the load on the network due to unnecessary page migration, it increases the load on the processor due to unnecessary polling of an unchanging synchronization variable (See [31]). The use of a full size page is also wasteful in that the entire `sentinel` structure can easily fit in a much smaller unit.

For the type of producer-consumer system of Figure 5, we can increase its efficiency several times by incorporating the ideas just presented into a new system. Two major modifications include:

[3]Therefore there is a difference here which we will refer to as *a page* and *a copy*; meaning a) the readable/writable page and b) just a copy of the page, respectively.

1. Use the *data driven* access mode. In data driven mode, a process can block on a memory read. The read is satisfied when another process performs an operation that causes a network refresh of the page. A network refresh is performed by the process possessing the consistent (writable) page and informs all the processes, throughout the network which have inconsistent copys of that page, to update their copy to match the consistent page.

2. Use *smaller pages.* The `sentinel` structure is smaller than 32 bytes, thus the programmer should use the METHERSHORT address space qualifier when allocating the structure. This means that the data units flowing over the network are only the first 32 bytes of each 8K page. Writing to other areas of the page is fine in the local environment, but the changes will not show up anywhere else in the network.

The modified program shown in Figure 6 implements these changes. The *data driven* option is accomplished through the use of the `metherpurge()` function, provided in the Mether library, which causes a *network refresh* of a page. The short pages are effected by giving the option `METHERSHORT1` to the `METHERMAPCLASS` macro as seen in lines 10-11 of Figure 6.

5.2 Synchronization Performance

Synchronization of multiple processes is an important component of any cooperative distributed computation. Synchronization should be efficient or performance will be poor for many applications, possibly to the point that sequential execution is preferable.

Mether provides efficient synchronization. Figure 7 presents the results of running a synchronization program using a conventional NSM (Mether in the strongly consistent mode, although the results apply as well to other NSMs) compared with the same program using the relaxed consistency modes provided by Mether.

The time axis is a log scale. This program is a synchronization "kernel" derived from analysis of multiprocessor applications designed for shared-memory multiprocessors. We use *short pages, application-controlled consistency* and the *memory-cycle-based synchronization* provided by Mether V3.

The curve for conventional memory (fully consistent) grows extremely rapidly with the number of processors. The latency curve for the Mether V3 interface is basically unvarying up to eight processes. Even for a small number of processors, the access time for shared variables grows rapidly. This same behavior can also be seen on other large systems with similar architectures and consistency requirements (See [13]).

Also shown in the graph is a plot for Mether-NFS, which uses a network file system to support Mether's applications model and libraries. All Mether semantics are supported on memory-mapped

Producer	Consumer
1 struct sentinel	1 struct sentinel
2 {	2 {
3 int gen, value;	3 int gen, value;
4 };	4 };
5 /* the SECOND Metherpage */	5 /* the SECOND Metherpage */
6 #define SECONDPAGE	6 #define SECONDPAGE
7 METHERBASE+METHERPAGESIZE	7 METHERBASE+METHERPAGESIZE
8 struct sentinel *sp, *readsp;	8 struct sentinel *sp, *readsp;
9 methersetup();	9 sp = methersetup();
10 sp = METHERMAPCLASS(METHERBASE,	10 sp = METHERMAPCLASS(SECONDPAGE,
11 METHERSHORT);	11 METHERSHORT);
12 readsp = METHERMAPCLASS(SECONDPAGE,	12 readsp = METHERMAPCLASS(METHERBASE,
13 METHERSHORT1);	13 METHERSHORT1);
14 sp->gen = 0;	14
15 RunConsumer();	15
16 sp->val = 0;	16
17	17
18 while(sp->val < 1000)	18 while(sp->val < 1000)
19 {	19 {
20 caddr_t purge = readsp;	20 caddr_t purge = readsp;
21 sp->val = sp->val + 1;	21 while (readsp->gen <= sp->gen)
22	22 ;
23 sp->gen = sp->gen + 1;	23
24 metherpurge(sp);	24
25 while (readsp->gen < sp->gen)	25 func(readsp->val);
26 ;	26 sp->gen++;
27	27 metherpurge(sp);
28 }	28 }
29 exit();	29 exit();

Figure 6: The Modified Producer and Consumer Processes

files. Performance is essentially linear to 16 processors.

5.2.1 Monte Carlo

The Monte Carlo program, described in [6], simulates a photon flux in a complex chamber with convex and concave surfaces[4]. The program can model, for example, laser radioisotope diffusion processes. A minimum of 37 million photons, with a chamber of 37 surfaces, are required to generate meaningful results.

The results are summarized in Figure 8. The fastest recorded execution time (for 37 million photons) was on a Cyber 205, hand-coded with portions in assembly, which ran in 30 minutes of

[4]This *photon flux* application models how photons react and diffuse within a chamber. Photons do not affect one another and, once they contact a surface, cease to exist.

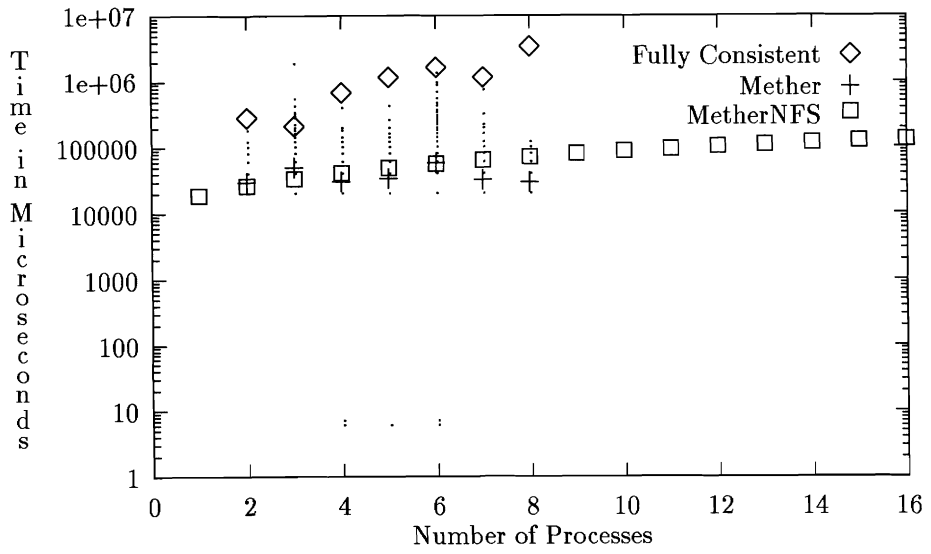


Figure 7: The synchronization problem

wall-clock time. On a Cray-2, a version written in C ran in 10 hours 20 minutes of wall-clock time, and used 3 hours and 20 minutes of cpu-seconds. Both of these applications ran in single-precision mode, which is equivalent to double-precision mode on a Sun. Identical code, except for modifications to use Mether network shared memory, was run in double precision on a network of 16 SparcStation ELCs and completed in 28 minutes wall clock time. Further measurements indicate that this relative performance advantage improves with increased problem size. The shared-memory model eased the port of this shared memory to the Suns and the efficiencies of the Mether model allowed effective use of the resources.

Architecture	Time
CDC Cyber 205	30 minutes
Cray 2	10 hours 20 minutes
Cray 2, 4 heads, vectorized (projected)	10 minutes
Cray C90, 16 heads, vectorized (projected)	1 minute
Sun Sparc ELC Farm (16 in NSM configuration)	28 minutes

Figure 8: Results for the monte carlo application

5.2.2 Sparse Matrix Solver

A second important application is a multiple-process sparse matrix solver[28]. Sparse matrix factorization is a computational kernel of almost any implicit finite element or finite difference code. For example, Spice and Nastran, when used on a large problem, will have run times dominated by sparse matrix factorization[34]. In addition, matrix operations are necessary for many numerical methods.

The sparse solver was designed and written for portability and used successfully on a number of parallel machines including an Intel iPSC2/VX. It was used in a parallel implementation of Pisces 2B, a device modelling program which uses a two dimensional finite difference model. The version of the program adapted for Mether is written in a variant of Fortran designed for a four-processor Cray-2[28].

Suitability for NSM This particular problem is a good fit for NSM. While this problem is computationally significant this implementation also stresses an NSM by requiring a variety of data access modes. The implementation has the processes running in several different stages, each of which makes different demands on the NSM[5]. In the early stages (initialization and assignment of work), the problem is divided up and the workers exchange information about their work assignments. The time for this transfer of messages must not be a significant part of the total problem time. In particular, the synchronization mechanisms discussed earlier must be efficient. The workers then each compute their assigned part of the matrix, eventually transmitting their partial results to a distinguished set of processes via a large shared array.

The program requires an NSM to support both a low-overhead, message-like communication via the memory and conventional coherent memory semantics at different times. The program uses these messages to synchronize while concurrently requiring the NSM to support a large, consistent array.

Performance Figure 9 shows the performance of the sparse solver for two dimensional arrays, ranging in size from 15 by 15 to 40 by 40 elements. The scaling achieved for Mether is comparable with the scaling achieved on a Cray-2 or an Intel hypercube[28]. Also notice there is an encouraging and expected pattern emerging which shows that for a fixed array size, doubling the number of processors from one to two to four has the effect of halving, and then halving again the process time.

[5] For details of the implementation, see [30].

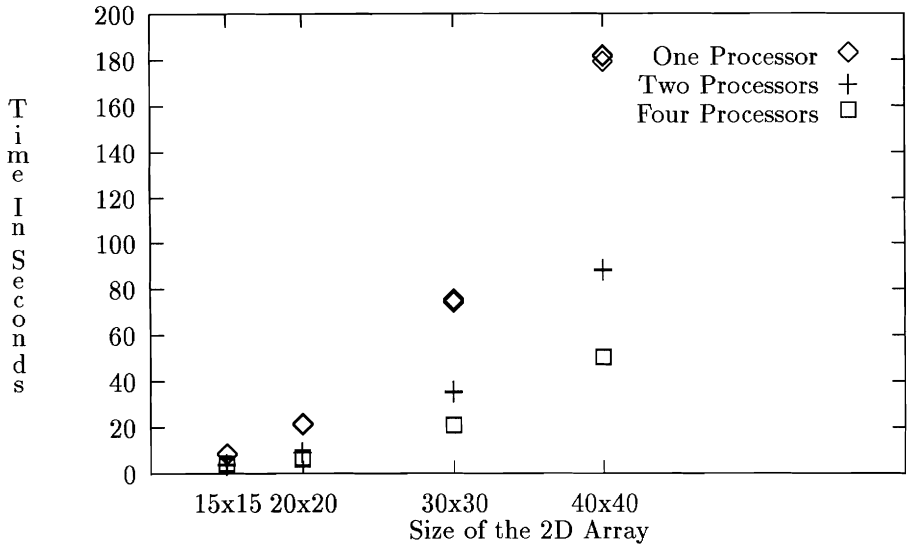


Figure 9: Performance of the sparse matrix solver for one, two, and four processes

5.2.3 DNA Pattern Matching

The third application is a DNA pattern matching algorithm, described in detail in [19].

The problem is as follows:

As DNA sequences are represented by a sequence of letters from the set {A,C,G,T} (for example, AGCTAAGAT...TAC)

We are given:

1. a *target* group of DNA sequences, and
2. a *key* DNA sequence,

One must select from the *target* group the sequence which most *closely* matches the *key* sequence. Note that the match need not be identical.

The DNA sequences are large enough that significant computation must be done to perform the selection. Fortunately the problem can be decomposed in such a way that an "embarassingly" parallel solution scheme can be used; this has allowed the problem to be run effectively on various parallel machines, including the massively parallel Splash[19] assembly of Field Programmable Gate Array (FPGA) devices.

The parallelization strategy used a process on each machine with each process capable of matching strings. One process also serves as a Problem Distributor. The problem distributer determines which set of strings should be matched, locates an available "matcher" process to perform the

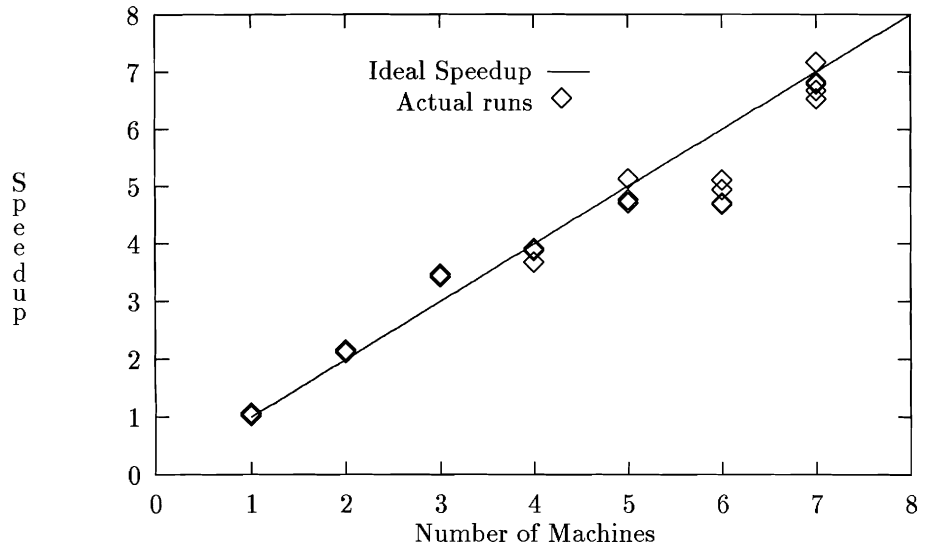


Figure 10: Sample DNA runs on 1 to 7 Sun 3s

match, and assigns the strings to it. The strings are stored in a large shared array, with pointers to the strings passed to the "matcher" processes.

Sample Runs on Sun 3s Figure 10 shows the execution results of highly optimized versions of the DNA matcher running on a farm of Sun 3s. There is a near-linear speedup for an increased number of processors. The deviations from the linear speedup line are due to differing processor performance. In fact, this application exhibits an unanticipated load balancing effect. Some of the newer, faster processors finish earlier and are given additional work to do, thus running more than their *fair share* of tasks if equivalent processors were used.

5.3 Discussion of Measured Application Performance

These performance results demonstrate that:

- Several types of shared-memory applications can run effectively in a networking environment using the Mether memory model.
- Supercomputer-level performance can be, and has been, achieved on these applications.

The factors affecting application performance are the page fault rate, the page fault service time ("latency"), and the "throughput" seen for data transfer. For each of the applications we studied, the lower the latency the faster they will complete, indicating that page fault rates and service times

are crucial to performance. Mether page fault service requires 100ms for long pages and 20ms for short pages; the data transfer rate can thus be calculated as 80 KB/s using a simple model where page faults drive the data transfer. (The Mether-NFS prototype requires 25 ms for a long page and 5ms for a short page, with measured throughput of about 320 KB/s). If the applications use a more sophisticated model for data transfer, taking advantage of some overlap in fault generation and fault service, the data transfer rate can be increased 200-300 KB/s. The basic page fault costs can be combined with invalidation traffic to determine additional latency due to this traffic, this reduces the need for page fault counter instruments.

Bandwidth is not crucial for the Monte Carlo problem unless the precomputed sine/cosine arrays are very large. For the sparse matrix solver, computation time increases as a cube of the size of the array and communication increases as the 1.9th power of the size of the matrix. Given the key role of synchronization, and since synchronization is done using one-word objects requiring a short page, the sensitivity to latency is much greater. The difference in fault service times between short pages and long pages is dominated by fragmentation/reassembly overhead for large pages rather than network bandwidth.

6 Conclusions and Future Work

We feel that a number of key conclusions about Network Shared Memories can now be drawn.

First, Network Shared Memories provide the programmer with a vehicle for transporting applications designed for shared memory parallel processors to networks of workstations. The experiences gathered porting applications from the Cray 2 processor to the workstation-based Mether system firmly demonstrate this fact. In addition, it should be noted that the application code used were for *real applications* and *not* contrived examples.

Second, Network Shared memories can provide many of these applications with a very attractive alternative to traditional parallel processing architectures, both in terms of price and in terms of performance. The three applications studied in this paper showed both attractive speedup curves and good absolute performance. A price/performance advantage over the Cray 2 of over 300 to 1 was demonstrated in one case.

Third, a number of optimizations to Network Shared Memories were proposed, implemented and evaluated. For some classes of applications, altering the consistency semantics of the memory can offer significant performance advantages without affecting correctness. Applications examples which are able to exploit the relaxed consistency requirements were demonstrated.

The research questions opened are numerous, and our intended future direction is as follows:

One. The Mether experiments were performed in a local area network setting and several

features of the network (e.g., broadcast and low latency) were exploited to Mether's advantage. A significant test of both the general applicability of NSM and the specific optimizations embedded in Mether will be the extension of the software to support Wide-Area Networks.

Two. Heterogeneity of workstation architectures and software is a significant impediment to the use of DSM as a distributed systems abstraction. This is especially true of a system like Mether, which offers features at a low-enough level of abstraction so that certain hardware features might be visible. Mether is being ported to an alternative workstation architecture, the IBM RISC System/6000, in order to explore issues of heterogeneity. The RISC System/6000 implementation also promises a test of DSM as an abstraction for very high speed networking.

Three. The shared memory environment our example applications were designed for is vastly different than a Network Shared Memory. Most obvious are the failure modes of a tightly coupled shared memory versus a NSM which has complex failure modes. In the case of same architecture (homogeneous) where, if one machine goes down, they all go down, fault-tolerant architectures using replicas or other redundancy and recovery strategies can be envisioned, but robust designs are considerably more difficult than might at first appear.

Four. There are a number of improvements in the implementation which are targeted at improving applications support. For example, better integration of Mether 3.0 with compiler support (e.g., to reduce the use of syntactically-confusing macros) will provide a more transparent applications environment.

References

[1] J. Archibald and J. Baer. An evaluation of cache coherence solutions in shared-bus multiprocessor. *ACM Transactions on Computer Systems*, February 1986.

[2] Henri Bal, J. G. Steiner, and Andy Tannenbaum. Experience with distributed programming in Orca. In *Proc. IEEE CS Int. Conf on Computer Languages*, pages 79–89, March 1990.

[3] Gordon Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[4] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings, 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–175, 1990.

[5] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[6] Patrick J. Burns and Daniel V. Pryor. Vector and parallel monte carlo radiative heat transfer simulation. *Numerical Heat Transfer*, 16:97–124, 1989.

[7] Ramon Caceres. Measurements of wide area internet traffic. Technical Report Technical Report UCB/CSD 89/550, Univ. California, Berkeley, Computer Science Division (EECS), December 1989.

[8] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The amber system: Parallel programming on a network of multiprocessors. Technical Report 89-04-01, University of Washington, Department of Computer Science and Engineering, September 1989.

[9] David R. Cheriton. Problem-oriented shared memory: A decentralized approach to distributed systems design. In *Proceedings of the Sixth IEEE Distributed Computing Systems Conference*, pages 190–197, 1986.

[10] D.R. Cheriton. The v kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[11] G. Delp, D. Farber, R. Minnich, J.M. Smith, and M. Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, July 1991.

[12] Gary S. Delp. The architecture and implementation of memnet : A high-speed shared memory computer communication network. Ph.d. thesis, Department of Electrical Engineering, University of Delaware, Newark, Delaware 19716, April 1988.

[13] Thomas M. Dunigan. Kendall square multiprocessor: Early experiences and performance. Technical report, Oak Ridge National Laboratory, 1992.

[14] Lenoski et. al. Design of the stanford dash multiprocessor. Technical report, Stanford University, December 1989.

[15] David J. Farber. The distributed computing system. In *Proceedings, 1973 COMPCON*. IEEE, 1973.

[16] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Usenix- Winter 89*, pages pp. 229–243. Usenix, February 1989.

[17] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for mach. Technical report, CMU, 1988.

[18] B.J. Walker G.J. Popek. Issues of network transparency and file replication in the distributed filesystem component of locus. Technical Report CSD830905, University of California, Los Angeles; Computer Science Department, 1983.

[19] Maya Gokhale, Bill Holmes, Andy Kopser, Dan Lopresti, and Ronald Minnich. Splash: A reconfigurable linear logic array. In *ICPP 90.* ICPP, 1990.

[20] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the Tenth IEEE Distributed Computing Systems Conference*, 1990.

[21] David J. Farber Ivan Ming-Chit Tam, Jonathan M. Smith. A taxonomy-based comparison of several distributed shared memory systems. *ACM Operating Systems Review*, 24(3):40–67, July 1990.

[22] H.T. Kung. Synchronized and asynchronous parallel algorithms for multiprocessors. In J.F. Traub, editor, *New Directions and Recent Results in Algorithms and Complexity.* Academic Press, 1978.

[23] K. Li and R. Schaefer. A hypercube shared virtual memory system. In *ICPP 89.* ICPP, 1989.

[24] Kai Li. Shared virtual memory on loosely coupled multiprocessors. Ph.d. thesis, Yale University, 1986.

[25] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM TOCS*, 7(4), November 1989.

[26] Don Libes. User-level shared variables. In *Usenix- Summer 85.* Usenix, 1985.

[27] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[28] Robert Francis Lucas. Solving planar systems of equations on distributed-memory multiprocessors. Technical report, Stanford University, 1987.

[29] D. Mills. The network time protocol. Technical report, University of Delaware, 1988.

[30] Ronald G. Minnich. Mether: A memory system for network multiprocessors. Ph.d. thesis, Dept. Comp. and Info. Sciences, University of Pennsylvania, 1990.

[31] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proceedings of the Tenth IEEE Distributed Computing Systems Conference*, 1990.

[32] D. L. Nelson and P.J. Leach. The architecture and applications of the apollo domain. *IEEE Computer Graphics*, pages 58–66, April 1984.

[33] U. Ramachandran and M. Y. A. Khalidi. An implementation of distributed shared memory. Technical report, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, December 1988.

[34] Albert C. Reynolds Richard L. Burden, J. Douglas Faires. *Numerical Analysis.* Prindle, Weber & Schmidt, Boston, Massachusetts, 2nd edition, 1981.

[35] Jonathan M. Smith and David J. Farber. Traffic characteristics of a distributed memory system. *Computer Networks and ISDN Systems*, 22(2):143–154, September 1991.

[36] Alfred Z. Spector. Performing remote operations efficiently on a local area network. *Communications of the ACM*, April 1982.

[37] Ivan Ming-Chit Tam and David J. Farber. Capnet - an alternative approach to ultra high speed networks. In *Proceedings, International Communication Conference*, 1990.

[38] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[39] L. Wittie and C. Maples. Merlin: Massively parallel heterogeneous computing. In *1989 International Conference on Parallel Processing*, pages 142–150, 1989.